Optimizing the Four-Index Integral Transform Using Data Movement Lower Bounds Analysis

Samyam Rajbhandari, Fabrice Rastello, Karol Kowalski, Sriram Krishnamoorthy, P. Sadayappan

Optimizing the Four-Index Integral Transform Using Data Movement Lower Bounds Analysis

Samyam Rajbhandari Fabrice Rastello [+] Karol Kowalski [*]
Sriram Krishnamoorthy [*] P. Sadayappan

The Ohio State University
INRIA [+]
Pacific Northwest National Laboratory [*]
rajbhandari.4@osu.edu fabrice.rastello@inria.fr Karol.Kowalski@pnnl.gov
sriram@pnnl.gov sadayappan.1@osu.edu

Abstract

The four-index integral transform is a fundamental and computationally demanding calculation used in many computational chemistry suites such as NWChem. It transforms a four-dimensional tensor from one basis to another. This transformation is most efficiently implemented as a sequence of four tensor contractions that each contract a four-dimensional tensor with a two-dimensional transformation matrix. Differing degrees of permutation symmetry in the intermediate and final tensors in the sequence of contractions cause intermediate tensors to be much larger than the final tensor and limit the number of electronic states in the modeled systems.

Loop fusion, in conjunction with tiling, can be very effective in reducing the total space requirement, as well as data movement. However, the large number of possible choices for loop fusion and tiling, and data/computation distribution across a parallel system, make it challenging to develop an optimized parallel implementation for the four-index integral transform. We develop a novel approach to address this problem, using lower bounds modeling of data movement complexity. We establish relationships between available aggregate physical memory in a parallel computer system and ineffective fusion configurations, enabling their pruning and consequent identification of effective choices and a characterization of optimality criteria. This work has resulted in the development of a significantly improved implementation of the four-index transform that enables higher performance and the ability to model larger electronic systems than the current implementation in the NWChem quantum chemistry software suite.

1. Introduction

The four-index integral transform is a computationally demanding calculation implemented in numerous computational chemistry suites, including NWChem [4], ACES [1], GAMESS [23], MOLPRO [3], MPQC [2], and PSI [5]. It is used to transform a four-dimensional tensor of coefficients (A), using four applications of a two-dimensional transformation matrix B. The computation can be specified as follows:

$$C[\alpha,\beta,\gamma,\delta] = \sum_{i,j,k,l} A[i,j,k,l] \cdot B[\alpha,i] \cdot B[\beta,j] \cdot B[\gamma,k] \cdot B[\delta,l] \quad (1)$$

However, efficient implementation of this transformation is non-trivial and has been the subject of a number of efforts [6, 8, 10, 11, 16, 19–21, 24–27]. The direct conversion of the above expression into code would result in an implementation with a computational complexity of $O(n^8)$, if all eight indices (α, β, γ, δ, i, j, k, l) range from 1 to n. By utilizing algebraic properties of associativity, commutativity, and distributivity, the transformation can be implemented with a computational complexity of $O(n^5)$ (as discussed in greater detail in the next section) as a sequence of four steps, each involving the tensor product of a 4D tensor with the 2D transformation matrix B. There are a large number of possible ways of partitioning the data and computation across a parallel system, along with fusion/tiling choices to reduce data movement through the memory hierarchy. Further complicating matters is the fact that the tensors have varying degrees of permutation symmetry (elaborated in the next section). It is critical to exploit the savings in space and computational operations by only keeping distinct elements, but this complicates reasoning about the data movement cost of alternative distributed implementations and the numerous fusion/tiling choices. A key challenge is that the intermediate tensors in the four-step index transformation are larger than the final result. This means that either judicious fusion across the steps is required to optimize memory (which complicates the implementation) or that memory use is suboptimal, limiting the size of the systems that can be modeled.

In this paper, we report on the development of a significantly improved parallel implementation of the four-index transform in NWChem. We pursue a novel approach using

data movement lower bounds to guide the optimization process. For the four-index transform, the combination of the number of fusion choices and tile sizes along the loops is prohibitively high. Each tensor contraction in the sequence of four contractions has five nested loops. For each adjacent pair of contractions, four of the five nested loops are common and any combination of these can be selected for fusion. It is also possible to fuse a subset of three common loops across a contiguous set of three contractions, or one of two common loops across all four contractions. Further, we must consider tiling choices for loops. The optimal configuration is not the same for different problem sizes, i.e., auto tuning will require execution of thousands of configurations for each problem size of interest. Thus, neither analytical model-based optimization, nor any successful autotuning approach has been previously reported for the four-index integral transform.

We avoid the combinatorial explosion in the number of modeled code configurations by using a novel and completely different approach towards performance modeling to guide the optimization: *by using lower bounds on data movement complexity*. Lower bounds on data movement (explained in the next section) are schedule-independent assertions on the minimal amount of data movement required for any valid implementation of a given computation (abstracted as a computational directed acyclic graph). This enables identification of effective fusion configurations, along with a characterization of optimality properties with respect to the largest problem sizes that can be transformed, as a function of the aggregate physical memory on the parallel computer system. The resulting implementation of the four-index transform significantly improves on currently implemented versions in NWChem.

The paper makes the following contributions:

- For sequences of loops representing a producer-consumer relationship, it develops the first approach to use data movement lower bounds to reason about the utility of loop fusion to reduce data movement **(Section 4)** and its application to find an optimal fusion choice for the four-index integral transform **(Section 5)**.
- It describes the first use of data movement lower bounds to reason about minimal space requirements for I/O-free computations for the four-index integral transform **(Section 6)**.
- It develops a new parallel implementation **(Section 7)** of the four-index integral transform that improves upon the current state-of-the-art implementations in production computational chemistry codes in two ways: 1) For a given amount of total collective physical memory in a cluster, it can run the provably largest four-index transform that does not use file I/O or redundant re-computation of intermediate tensors; 2) For a given amount of per-process local memory, it minimizes the data movement between non-local and local memory.
- It presents an experimental evaluation of the new parallel implementation, demonstrating significant performance improvement for large problems, and the ability to run larger problems than previously possible with the production NWChem computational chemistry suite **(Section 8)**. For example, we show a practical problem that requires more than 12 Terabytes of collective global memory to execute without fusion. Using our implementation, we execute this problem on a cluster with less than 9 Terabytes of collective global memory.

2. Background and Related Work

In this section, we provide necessary background information before describing the new contributions. We first elaborate on the four-index integral transform and its implementations in NWChem, followed by a brief introduction to data movement lower bounds.

2.1 Four-Index Integral Transform

In equation 1, if all indices $(\alpha, \beta, \gamma, \delta, i, j, k, l)$ range over the integer interval $[1, n]$, a naive implementation of the transform would take $5n^8$ operations. In practice, it is implemented as a sequence of four tensor contractions (shown in Equation 2). Each tensor contraction (higher dimensional analogs of matrix-matrix product) has a computational cost of $2n^5$ operations. The output tensor of a contraction serves as an input tensor for subsequent computation.

$$
\begin{aligned}
O1[\alpha, j, k, l] &= \sum_i A[i, j, k, l] \cdot B[\alpha, i] \\
O2[\alpha, \beta, k, l] &= \sum_j O1[\alpha, j, k, l] \cdot B[\beta, j] \\
O3[\alpha, \beta, \gamma, l] &= \sum_k O2[\alpha, \beta, k, l] \cdot B[\gamma, k] \\
C[\alpha, \beta, \gamma, \delta] &= \sum_l O3[\alpha, \beta, \gamma, l] \cdot B[\delta, l]
\end{aligned}
\tag{2}
$$

It can be seen that for each adjacent pair of tensor contractions, four of the five loops iterate over common tensor indices, while one loop differs. Thus, many possible fusion choices exist, where loops over the same tensor indices are fused across two or more consecutive contractions.

Symmetry in tensors. Tensors in the four-index transform exhibit permutation symmetry. A tensor is symmetric with respect to a subset of its indices if permuting the indices within the subset does not change the value of the tensor. As an example, consider a tensor $V[a, b, i, j]$. We say that indices a and b are symmetric if $V[a, b, i, j] = V[b, a, i, j]$[1]. A symmetric tensor can have multiple symmetry groups. For example, a, b and i, j are two symmetry groups for V if $V[a, b, i, j] = V[b, a, i, j] = V[b, a, j, i] = V[a, b, j, i]$. This tensor can be represented compactly by only explicitly storing elements $V[a < b, i < j]$ due to the symmetry, denoted $V[ab, ij]$. The symmetry properties imply that only a fraction $(1/d!)$ of the elements of the full tensor need to be

[1] Tensors in quantum chemistry generally possess anti-symmetry rather than symmetry, i.e., $V[a, b, i, j] = -V[b, a, i, j]$, and our codes actually incorporate anti-symmetry. However, for simplicity, we use symmetric tensors in our presentation.

```
/*Memory required: 3n^4/4,
  Computation: 4n^5*/
malloc(A,n^4/4);
malloc(O1,n^4/2)
for a, i, j, k>l
 O1[a,j,k>l] +=
  A[i>j,k>l]*B[a,i] //n^5
delete(A);malloc(O2,n^4/4)
for a>b, j, k>l
 O2[a>b,k>l] +=
  O1[a,j,k>l]*B[b,j]//n^5
delete(O1);malloc(O3,n^4/2)
for a>b, c, k, l
 O3[a>b,c,l] +=
  O2[a>b,k>l]*B[c,k]  //n^5
delete(O3);malloc(C,n^4/32)
for a>b, c>d, l
 C[a>b,c>d] +=
  O3[a>b,c,l]*B[d,j]  //n^5
```

Listing 1: Unfused

```
/*Memory required: n^4/2
  Computation: 4n^5 */
malloc(A,n^4/4);malloc(O1,n^2)
malloc(O2,n^4/4)
for (k>l)
 for (a,i,j)
  O1[a,j]+=A[i>j,k>l]*B[a,i] //n^5
 for (a>b,j)
  O2[a>b,k>l]+=O1[a,j]*B[b,j]//n^5
delete(A); delete(O1)
malloc(O3,n^2); malloc(C,n^4/32)
for (a>b)
 for (c,k,l)
  O3[a>b,c,l] +=
   O2[a>b,k>l]*B[c,k] //n^5
 for (c>d,l)
  C[a>b,c>d] +=
   O3[a>b,c,l]*B[d,j] //n^5
```

Listing 2: First two and last two fused

```
/*Memory required: n^3
  Computation: n^6+ */
malloc(C,n^4/32)
for (a>b)
 malloc(O1,n^3)
 for (i,j,k>l)
  O1[j,k>l] +=
   A[i>j,k>l])*B[a,i]//n^6
 malloc(O2,n^2)
 for (j,k>l)
  O2[k>l]+=O1[j,k>l]*B[b,j]//n^5
 delete(O1); malloc(O3,n)
 for (c,k,l)
  O3[l]+=O2[k>l]*B[c,k]//n^5
 delete(O2,n^2)
 for (c,d,l)
  C[a>b,c>d]+=O3[l]*B[d,j]//n^5
```

Listing 3: With re-computation

Three implementation variants for 4-index transform

Table 1: Sizes of intermediate tensors. s is the factor of storage reduction due to spatial symmetry in tensor C.

Tensors	Sizes
A	$n^4/4$
O1	$n^4/2$
O2	$n^4/4$
O3	$n^4/2$
C	$n^4/(4s)$

```
1  int node_id = GA_Nodeid()
2  for l, k, j, alpha
3    if(node_id owns tile C[alpha,j,k,l])
4      malloc (C_buf, Tilesize)
5      for i
6        A_buf = GA_Get(Tile A[i,j,k,l])
7        B_buf = GA_Get(Tile B[alpha,i])
8        DGEMM(A_buf, B_buf, C_buf)
9      GA_Put(C_buf to tile C[alpha,j,k,l])
```

Listing 4: Implementation of the tensor contraction `C[alpha,j,k,l] += A[i,j,k,l] . B[alpha,i]` in NWChem

stored in the memory, where d is the number of dimensions in a symmetry group. In the above example, V has two symmetry groups of size 2, so that only a fraction $\frac{1}{2} \times \frac{1}{2}$ of the elements need be explicitly stored.

Permutation symmetry in the tensors in equation 2 can be represented as $A[ij,kl], O1[\alpha,j,kl], O2[\alpha\beta,kl], O3[\alpha\beta,\gamma,l]$ and $C[\alpha\beta,\gamma\delta]$. In addition to permutation symmetry, the final result tensor in molecular basis can also exhibit *spatial symmetry*. Spatial symmetry is a structured form of sparsity that causes certain blocks to be zero if the index ranges for that block (corresponding to specific molecular orbitals) satisfy some conditions. Whereas permutation symmetry reduces the size of all tensors—input, intermediate, and output—in the four-index transform, spatial symmetry results in additional size reduction only for the final output tensor C. The sizes of various tensors for the four-index transform are shown in Table 1.

Tensor data structures in NWChem. In NWChem, tensors are blocked along each dimension, resulting in a set of data-tiles representing the tensor. These data-tiles are linearized and distributed using Global Arrays [18], a global address space framework that provides a logical shared view of arrays in a distributed-memory parallel system. Each parallel process in a parallel program can *get*, *put*, or additively *update* any arbitrary data-tile of the tensor.

During a contraction, each distributed processor node is responsible for computing a subset of the tensor tiles. To this end, each processor uses `GA_Get` requests for the input data-tiles required to compute each of its assigned output data-tiles. After accumulating all contributions to an output data-tile, the processor then stores it using `GA_Put`.

Listing 4 depicts the implementation of the first tensor contraction in the four-index transform as performed in NWChem. Here, loop indices represent tile indices in different tensor dimensions. Line 3 checks if the current output-tile is the responsibility of the process. If so, lines 6 and 7 request the required input data-tiles using `GA_Get`. Line 9 stores the computed result using `GA_Put`.

2.2 Related Prior Work

Fusing sequences of tensor contractions to minimize memory use or data movement has been studied using a variety of approaches. Lam et al. [15] viewed individual tensor contractions as loop nests that constitute expression trees and determine the execution order and fusion configuration that minimizes memory used for intermediates, indirectly optimizing data locality. Gao et al. [12] presented an approach to prune the space of candidate fusion and tiling configurations to minimize disk I/O cost. The approach involves computing the disk I/O cost for every valid loop structure

for a given sequence of expressions, which grows exponentially with the number of contractions being fused. Sahoo et al. [22] pruned the search space to be explored by casting a part of the choice of fusion configurations as a tile size selection problem, which is then solved using a non-linear optimization solver. Ma et al. [17] extend this approach to deal with multi-level memory hierarchies and further simplify the space by considering only the first-order approximation of the costs. None of these frameworks handled the permutation symmetry in four-index transform. In this paper, we improve upon these approaches by employing lower-bounds analysis on dat amovement costs to identify effective configurations while also handling permutation symmetry.

Over the years, many schemes for the four-index transform have been devised [6, 8, 10, 11, 16, 19–21, 24–27]. These approaches are based on heuristics and experience rather than concrete cost models. Many of these variants have been implemented in the NWChem software suite. the following are the most widely used and performant implementations of four-index transform in NWChem:

Unfused: Fully unfused version, where A, $O1$, $O2$, $O3$, and C tensors are fully created. This form is computationally the most efficient, with an $O(n^5)$ scaling of the algorithm but requires more than $\frac{3n^4}{4}$ words of global memory to fully store the input and output of the largest tensor contraction in the 4-index transform: ($|O1| + |O2|$). This algorithm is shown in Listing 1.

Fused 12-34: Loop fusion allows significant reduction in the space required for intermediate tensors, since the index corresponding to any common loop over a producer and consumer does not need a corresponding explicit dimension in the tensor – sequential computing over that index is feasible and a lower dimensional intermediate tensor can be repeatedly reused. This algorithm is shown in Listing 2. It can be seen that the previously dominant $O1$ and $O3$ intermediates are now lower dimensional tensors and no longer very demanding of memory. This algorithm requires at least $\frac{n^4}{2}$ words of global memory.

Recompute: This option allows a significant reduction in the memory footprint, at the cost of redundant computation. Generally, this option will be more time consuming but has the lowest memory requirement. The factor of reduction in the global memory required is proportional to the amount of re-computation. Listing 3 shows such an algorithm.

2.3 Data movement lower bounds

Consider matrix-matrix multiplication

$$C[i,j] = A[i,k] \times B[k,j] \tag{3}$$

The standard $O(N^3)$ algorithm is shown in two forms in Figure 1: (a) basic untiled form and (b) tiled form. Both versions require the same number ($2N^3$) of arithmetic operations to compute. However, they differ in the amount of data movement in a hierarchical memory system. Let us consider the simple memory hierarchy containing two levels, with all matrices fitting within the latter level, but too large to fit in the level closer to the processor, which can store only S data elements, $S < N^2$. The untiled version requires the entire matrix B to be accessed for each iteration of the outer i-loop. Therefore, the number of data movement operations from the latter level of memory to the closer level for the untiled version is at least N^3 (ignoring data movement for A and C). However, the tiled version needs much less data movement, $O(\frac{N^3}{T})$, for $T < \sqrt{\frac{S}{3}}$, where T is the tile size.

```
for (i = 0; i <= ni-1; i++)
  for (j = 0; j <= nj-1; j++)
    for (k = 0; k <= nk-1; ++k)
      C[i][j]+=A[i][k]*B[k][j];
```

```
for (ti=0; ti<=floord((ni-1), T); ti++)
  for (tj=0; tj<=floord((nj-1), T); tj++)
    for (tk=0; tk<=floord((nk-1), T); tk++)
      for (i=T*ti; i<min((T*(ti+1)),ni); i++)
        for (j=T*tj; j<min((T*(ti+1)),nj); j++)
          for (k=T*tk; k<min((T*(ti+1)),nk); k++)
            C[i][j]+=A[i][k]*B[k][j];
```

Figure 1: Matrix-matrix multiplication kernel: untiled (left) and tiled (right)

The example shows that unlike the invariance of computational complexity across the two versions ($2N^3$ operations irrespective of the values of S and T), the amount of data movement can depend on: i) which version of the code is used, ii) the size S of fast memory, and iii) the tile size T. A question of interest is: *What is the minimum possible amount of data movement among all equivalent forms of code, to compute the standard matrix-matrix multiplication algorithm?*

In general, this question cannot be answered without examining a combinatorially explosive number of possible valid schedules for a given computation. The foundational work by Hong and Kung [13] developed an approach to provide lower bounds on the data movement required by any valid execution of the set of operations for a given algorithm. They showed that on a system with a two-level memory hierarchy with fast memory capacity of S elements, the minimum I/O for multiplying two $n \times n$ matrices is $\Omega(n^3/\sqrt{S})$. Irony et al. [14] provided a lower bound with scaling constants for multiplication of an $n_i \times n_j$ matrix and an $n_j \times n_k$ matrix to be $\Omega(n_i \times n_j \times n_k/(2\sqrt{2S})$. A much tighter bound was provided by Dongarra et al. [9]: $\Omega(1.73 n_i \times n_j \times n_k/\sqrt{S})$.

Tensor contractions represent a generalization of matrix-matrix multiplication. We use the lower bound complexity results for matrix-matrix multiplication in exploring fusion/tiling choices for the four-index integral transform computation.

3. Overview of Approach

The goal of this paper is to construct a distributed four-index transform schedule that optimizes two aspects: i) minimize inter-node communication between distributed memory nodes, and ii) maximize the problem size that can be executed for a given amount of collective physical memory, without going to disk – important because nodes in super-

computers often do not have local disks and the collective bandwidth to the file system disks is very low. Proving the optimality of an algorithm is important because it guarantees that we have arrived at the best possible solution and no further improvement is feasible.

Reducing communication volume between distributed memory nodes, and eliminating disk I/O corresponds to reducing data movement between slow and fast memory at different levels of the memory hierarchy. We use fusion as a tool to reduce the data movement in the 4-index transform. To arrive at the final solution, we develop lower bounds guided fusion analysis. Our lower bound analysis quantifies the maximum possible reduction in data movement via fusion. If the maximum reduction possible is a small fraction of the total data movement of the unfused schedule, then we know that fusion is not very useful and such configurations can be eliminated from consideration. On the other hand if the reduction in data movement is significant, then fusion is useful, and an effort should be made towards finding such a fused schedule. Identifying the utility of fusion is described in detail in Section 4.

In Section 5, we apply the lower bound analysis developed in Section 4 to the 4-index transform and identify the minimum possible data movement for various fusion choices. By quantifying the minimum data movement (or the maximum reduction in data movement) for a fusion strategy, we focus on creating schedules for only those fusion choices that have potential to yield significant reduction in data movement. In section 6, we present a schedule based on an optimal fusion choice that maximizes the reduction in data movement, thus achieving the minimum data movement possible between slow and fast memory.

Our schedule achieves the minimum possible data movement only when the size of the final output tensor fits in fast memory. The next logical question is "Is this schedule optimal in terms of problem size?". In other words, is it possible to create other schedules that might minimize the data movement even when the final output does not fit in fast memory. To answer this question, we derive the necessary conditions for achieving the minimum data movement. We show that it is necessary for the fast memory to be larger than the size of the output to achieve the minimum possible data movement. In other words, if the output size is larger than the size of the fast memory then there can be no schedules that achieve the data movement lower bound. This necessary condition is derived in Section 6.

At the highest level of the memory hierarchy, where disk is the slow memory and collective global memory is the fast memory, the necessary condition described above implies that our schedule maximizes the problem size that can be executed on a given amount of collective physical memory without going to disk. At the next level of memory hierarchy, where global collective memory is the slow memory and local memory is the fast memory, our fusion strategy minimizes communication between local memory and global memory. These details, along with other implementation details are provided in Section 7.

4. Utility of Fusion

In this section, we elaborate on how analysis of data movement lower bounds can be useful in developing high-performance code for the four-index integral transform. As explained earlier, the four-index transform is computed as a sequence of four tensor contractions, with the output tensor from an immediately preceding contraction serving as an input tensor for the immediately following contraction. There are many possible choices of loop permutation and loop fusion, and for each choice of loop permutation/fusion, there is a very large search space of tile sizes for tiled execution (since all loops of a tensor contraction are fully permutable, they are all fully tileable too).

As an aid to eliminating many possible fusion choices and identifying the most promising ones, we develop a necessary condition for the utility of fusion in the context of general sequence of two producer-consumer computations. We first formalize the notion of fusion of computations.

Definition 4.1 (Fusion)**.** *Let $op1$ and $op2$ be two computations where the output of $op1$ is consumed by $op2$. A non-fused schedule of $op1 \cup op2$ is a schedule where all operations constituting $op1$ are fully completed before $op2$ starts. A fused schedule is a schedule where some interleaving of the constituent elementary operations of $op1$ and $op2$ occur.*

Next, we state the Fusion Lemma, which bounds the potential benefit from fusion in terms of the size of the output data from the producer computation used by the consumer computation, and the inherent minimal data movement (we interchangeably use the term I/O for data movement in a memory hierarchy, following the terminology of Hong & Kung [13]) complexity of the producer and consumer computations if executed separately.

Lemma 4.2 (Fusion Lemma)**.** *Let computation $C1$ and computation $C2$ represent a producer-consumer pair, where the output $O1$ produced by $C1$ forms a subset of the input $I2$ of $C2$. Let $IO_{LB}(C1)$ and $IO_{LB}(C2)$ represent known I/O lower bounds for $C1$ and $C2$, respectively. Consider the fused computation $C12$ fusing $C1$ and $C2$, with each output data element from $C1$ serving as an input data element for $C2$. If no element of the output set $O1$ is used internally for any computation in $C1$, any valid schedule for $C12$ has an I/O lower bound given by: $IO_{LB}(C12) = IO_{LB}(C1) + IO_{LB}(C2) - 2 * |O1|$.*

We omit the formal proof of this lemma in the main body of the paper and provide it in Appendix A. The key insight from the Fusion Lemma is that the potential benefit from performing fusion across two computations is dependent on the size of the intermediate data (produced by the first computation and consumed by the second) relative to the "inherent" data movement complexity of the two computations. If the minimal data movement required for each of the producer/-consumer computations is much larger than the size of the intermediate data, fusion is futile since the potential reduction in data volume is bounded by twice the size of the intermediate data. On the other hand, if the intermediate data

volume is much larger than the "inherent" data movement requirements for the individual computations, fusion can be very beneficial in reducing data movement.

We illustrate the Fusion Lemma using the example of a sequence of two matrix products: C = AB; E = CD. The output matrix C produced by the first matrix product is used as an input for the second matrix product. Let us first consider the case where all matrices are of the same size, $N \times N$, with the matrices being larger than the cache capacity S. The tightest published lower bound on data movement for matrix multiplication (by Dongarra et al. [9]) is $\Omega(1.73N^3/\sqrt{S})$. An efficient tiled implementation achieves a data movement (for the highest order term) of $2N^3/\sqrt{S}$. From the Fusion Lemma, the minimum possible I/O for any fused algorithm is $2 \times 1.73N^3/\sqrt{S} - 2N^2$. Since efficiently tiled but unfused execution of the sequence of two matrix products can be executed with an I/O cost of $2 \times 2N^3/\sqrt{S}$, the maximum possible reduction in I/O from fusion is limited to be less than $0.54N^3/\sqrt{S} + 2N^2$. Fractionally, the reduction is upper-bounded to be under 0.54/2, or around 27% (ignoring the lower order N^2 term).

Next let us consider a non-square case where A and B are $N \times K$ and $K \times N$, respectively, and D is also $N \times K$, with $N \gg K$. Now, the 'inherent" I/O complexity for each of the two producer/consumer operations is $\Omega(1.73N^2K/\sqrt{S})$, which can be much smaller than N^2. In this case, fusion of the sequence of matrix multiplications can be very beneficial in reducing the total volume of data movement.

5. Analysis: Four-Index Transform

In this section, we develop conditions for establishing tight bounds on I/O for some fusion configurations for the four-index transform.

5.1 Necessary Condition: Utility of Fusion

Consider the four-index transform re-written as shown in Equation 2. Let the extent of the tensor along each index be n. The first two contractions are:

$$O1[\alpha,j,k,l] = \sum_i A[i,j,k,l] \cdot B[\alpha,i] \quad (4)$$

$$O2[\alpha,\beta,k,l] = \sum_j O1[\alpha,j,k,l] \cdot B[\beta,j]$$

Each tensor contraction can be represented as a matrix multiplication (seeing resp. (j,k,l) / (α,k,l) as a "macro-index", ranging over the cross-product of multiple tensor indices in A / $O1$). Thus, each of the contractions in the sequence for the 4-index integral transformation is logically equivalent to a matrix product between a $n^3 \times n$ matrix and a $n \times n$ matrix. We can use the Fusion Lemma to derive conclusions on which combinations of fusion are beneficial. We initially ignore any symmetry, considering all matrices and tensors to be full and dense. However, the final results are based on considering symmetry.

From the result of Dongarra et al. [9], the I/O lower bound for the tensor contraction, i.e., matrix multiplication with the "macro" indices, is $1.73n^5/\sqrt{S}$ if S is sufficiently small. But if S is very large, this bound is lower than the sum of data

```
1  //  C[alpha,j,k,l] += A[i,j,k,l] . B[alpha,i]
2  //Does I/O equal to |C|+|A|+|B| if S >= n^2 + n + 1
3  load B[alpha*,i*]   //requires size n^2
4  for l, k, j
5    copy (A_buf[i*], A[i*,j,k,l]) //requires size n
6    for alpha
7      scalar c //allocate a scalar
8      for i
9        c += A_buf[i] * B[alpha,i]
10     C[alpha,j,k,l] += c
```

Listing 5: Tensor contraction schedule that achieves I/O equal to sum of its input and outputs

sizes of input and output tensors, which is $n^4 + O(n^2) + n^4$ = $2n^4$ if the lower order $O(n^2)$ term is ignored. Therefore, the I/O lower bound for one tensor contraction in the four-index transform sequence is $max(1.73n^5/\sqrt{S}, 2n^4)$. Considering a sequence of two consecutive tensor contractions from the four-index transform, the size of the intermediate tensor (output from the first contraction that is an input to the second) is n^4 elements. From the Fusion Lemma, an I/O lower bound for any fused schedule is $2 \times max(1.73n^5/\sqrt{S}, 2n^4) - 2n^4$. When S is very small, the $1.73n^5/\sqrt{S}$ term in the max expression is larger than $2n^4$, and the the I/O lower bound for any fused schedule is $3.46n^5/\sqrt{S}$, which is larger than the twice the size of the intermediate tensor if $3.46n^5/\sqrt{S} > 2n^4$, i.e., $S < 3n^2$. Thus, if the size of fast memory is much less than $3n^2$, the Fusion Lemma indicates that fusion would not be worthwhile.

Next let us consider the case where S is larger than n^2. In this case, the I/O lower bound from the Fusion Lemma, $2 \times max(1.73n^5/\sqrt{S}, 2n^4) - 2n^4$ = $4n^4 - 2n^4 = 2n^4$, half the minimal unfused I/O cost of $4n^4$ (sum of sizes of input and output tensor for each of the two contractions). Thus, when $S > n^2$, reduction in I/O from fusion is not ruled out by the Fusion Lemma, and we consider that scenario next.

5.2 Tight I/O Bound for Useful Fusion

In this subsection, we identify a tight I/O bound for fusion of two consecutive contractions in the four-index transform when $S \geq 3n^2 + n + 1$. The result are summarized in the following theorem.

Theorem 5.1 (Utility of fusing 2 contractions)**.** *Consider any consecutive pair of tensor contractions op*1 *and op*2 *in the four-index transform, where the extent of each tensor dimension is n. Let $A/O2$ be the input/output of op*1/*op*2*. Then, fusion is useful if $S \geq 3n^2 + n + 1$, with a tight I/O lower bound of $IO_{opt}(op1 \cup op2) = |A| + |O2|$ for the fused schedule.*

Proof. Without loss of generality let us use the first two contractions in the four-index transform (see Eq. 4) to represent any pair of consecutive contractions. We prove the theorem in two steps. First, we show that $|A| + |O2|$ is a lower bound on I/O for a fused schedule when S is sufficiently large. Next, we show that we can construct a fused schedule that achieves this lower bound when $S \geq 3n^2 + n + 1$.

```
//I1[alpha,j,k,l] += A[i,j,k,l] . B1[alpha,i]
//C[alpha,beta,k,l] += I1[alpha,j,k,l] . B2[beta,j]
//if S >= 3n^2 + n + 1
//Total I/O of |C|+|B0|+|B1|+|A|
//requires 2n^2 memory
load B1[alpha*,i*], B2[beta*,j*]
for l, k
  malloc (I1_buf, n^2)
  for j
    malloc (A_buf, n)
    copy (A_buf, A[i*,j,k,l])
    for alpha, i
      I1_buf[alpha,j] += A_buf[i] * B1[alpha,i]
  for beta, alpha, j
    C[alpha,beta,k,l] += I1_buf[alpha,j]*B2[beta,j]
```

Listing 6: Fused pair of tensor contractions that achieves I/O equal to sum of size of its input and outputs

When $S \geq n^2 + n + 1$, a tight I/O lower bound on individual tensor contractions in the four-index transform is given by the sum of the sizes of the input and output tensors. As each input tensor must be read at least once and the output tensor must be written at least once, this is a lower bound on I/O for the tensor contraction. Listing 5 achieves this I/O. Therefore, it is a tight bound. Now using Lemma 4.2, the lower bound on I/O for a fused schedule is given by:

$$(|A| + |O1|) + (|O1| + |O2|) - 2|O1| = |A| + |O2| \tag{5}$$

Additionally, when $S \geq 3n^2 + n + 1$, a schedule can be constructed that achieves this bound. Loops k and l can be fused: for any fixed value of k and l, the n^2 values of $A[i, j, k, l]$ can be initially loaded, allowing I/O-free computation of the n^2 values of $O1[\alpha, j, k, l]$. This, in turn, allows I/O-free computation of the n^2 values of $O2[\alpha, \beta, k, l]$, which are then stored. The I/O for such a schedule (shown in Listing 6) is exactly the sum of sizes of inputs and outputs (there is no intermediate I/O in executing the operations). Therefore, $|A| + |O2|$ is a tight I/O bound. □

5.3 Fusion Choices: Four-Index Transform

The optimal fusion strategy is one that minimizes the I/O between slow and fast memory. Different unique ways to fuse the four-index transform are as follows: 1. No fusion; 2. Fuse the first two and last two; 3. Fuse only a single pair; 4. Fuse three contractions; 5. Fuse all four contractions. This section's goal, as summarized in the following frame, is to identify which strategy is the best, i.e., the one with the least I/O.

The I/O lower bound for each of the fusion choices can be computed using Lemma 4.2, in conjunction with the tight I/O bound for individual tensor contractions and the result from Theorem 5.1. We already know that the tight I/O lower bound for each individual (or pair of) contractions is the sum of sizes of its input and output. For fusion of three or more contractions, the sum of its input and output sizes is a correct lower bound but not necessarily a tight bound. Below, we summarize the I/O bounds for each fusion choice.

We use the following notation: $op1234$ (resp., $op12\not34$, $op123\not4$, etc.) denotes fused computation of all tensor contractions (respectively, first two and last two with forced spilling in between, first three, etc.). Thus (using the sets as in Equation 2):

$$IO_{opt}(op1\not2\not3\not4) = |A|+|O1| + |O1|+|O2| + |O2|+|O3| + |O3|+|C|$$
$$IO_{opt}(op12\not34) = |A|+|O2| + |O2|+|C|$$
$$IO_{opt}(op1\not23\not4) = |A|+|O1| + |O1|+|O3| + |O3|+|C|$$
$$IO_{opt}(op123\not4) \geq |A|+|O3| + |O3|+|C|$$
$$IO_{opt}(op1234) \geq |A|+|C|$$

Thus far, we have assumed that the size of all $4D$ tensors is n^4. Hence $IO_{opt}(op123\not4) \geq IO_{opt}(op12\not34)$. However, in reality, $IO_{opt}(op123\not4) > IO_{opt}(op12\not34)$ because $|O3|$ is bigger than $|O2|$ due to symmetry (see Table 1). In addition, we also have trivially that $IO_{opt}(op1234) \leq IO_{opt}(op12\not34)$. Thus, we get the following total order.

Theorem 5.2 (Order of Fusions)**.** *Consider four consecutive tensor contractions op1, op2, op3, and op4 and the previously defined notation for fusion. Let $S = \Omega(n^2)$. Then,*

$$IO_{opt}(op1234) \leq IO_{opt}(op12\not34) < IO_{opt}(op123\not4)$$

In other words, the lower bounds show that fusion of the first two and last two contractions has lower I/O than no fusion, while fusing three contractions does not lower the I/O further. The only way it may be possible to reduce the I/O further is to fuse all four contractions together to achieve I/O of $(|A|+|C|)$, via full reuse of intermediates.

6. Necessary and Sufficient Conditions for Full Reuse

In this section, we develop necessary and sufficient conditions for full reuse of all intermediates in the four-index transform, resulting in an I/O of $|A| + |C|$. If the size of fast memory (S) is larger than the sum of all the intermediates in the four-index transform, then achieving an I/O of $|A| + |C|$ is trivial. Even a schedule that performs each of the tensor contractions individually without any fusion will achieve this I/O. However, when the intermediates are too big to fit in fast memory, is full reuse of intermediates still possible? If so, under what conditions?

We show that $S \geq |C|$ is a necessary and sufficient condition for a tight I/O bound of $(|A|+|C|)$.

6.1 Necessary Conditions

In this subsection, we show that if $IO_{opt}(op1234) = |A|+|C|$, then $S \geq |C|$. We first consider the case without symmetries and then show that the proof holds even with symmetric tensors.

Theorem 6.1 (No symmetry)**.** *Let $op1234$ represent the total computation described by Equation 2.*
Then, $IO_{opt}(op1234) = |A|+|C|$
only if $S \geq \min\left(|A|, |O1|, |O2|, |O3|, |C|\right)$

Proof. To achieve the I/O lower bound $(|A|+|C|)$, the intermediates have to be fully reused without any intervening

I/O in between the operations. Let $C[\alpha, \beta, \gamma, \delta]$ be the first computed element of the output, and the corresponding time stamp right before its computation be T_f. We call the *live set* (also denoted W) the already loaded values of A, partially or fully computed values of $O1$, $O2$, $O3$ and C that are to be used later on. We prove that this live set is at least of size $|C|$. Let $F0$, $F1$, $F2$, $F3$ and $F4$ be the already loaded/computed values of A, $O1$, $O2$, $O3$ and C, respectively (which may or may not be used later on). Let $D0, \ldots, D4$, be the set of values in $F0, \ldots, F4$, respectively, that will not be used later on. Therefore, the size of the live set $|W|$ is given by

$$(|F0| \dashv |D0|) + (|F1| \dashv |D1|) + (|F2| \dashv |D2|) + (|F3| \dashv |D3|) + (|F4| \dashv |D4|)$$

Note that to be spill-free (i.e., without any intervening I/O in between contractions), a schedule must be such that $|W| \leq S$. Indeed, W corresponds to simultaneously live variables: if all of them cannot fit in collective physical memory, it means some need to be stored on disk and retrieved later. We now establish a lower-bound on $|W|$.

Each element of C depends on all values of tensor A. At time step T_f, none of the output elements is yet fully computed. Therefore, at time step T_f, $F0 = A$ and $D4 = \emptyset$. $|W|$ simplifies to

$$(|A| \dashv |D0|) + (|F1| \dashv |D1|) + (|F2| \dashv |D2|) + (|F3| \dashv |D3|) + (|F4|) \quad (6)$$

Let us recall the computation of the first tensor contraction:

$$O1[\alpha, j, k, l] = A[i, j, k, l] * B0[\alpha, i]$$

Two key observations are as follows:
1. If a given $A[i, j, k, l]$ is in $D0$, then all $O1[*, j, k, l]$ must be in $F1$;
2. If $D0_{j,k,l}$ denotes all the $A[*, j, k, l]$ (at most n_i elements) in D0 and $F1_{j,k,l}$ denotes all the $O1[*, j, k, l]$ (exactly n_α elements) in $F1$, then $|F1_{j,k,l}|/|D0_{j,k,l}| \geq n_\alpha/n_i$.

Thus, $|F1| \geq |D0| \times n_\alpha/n_i = |D0| \times |O1|/|A|$. Applying the same reasoning on the remaining contractions leads to the following lower bound for $|W|$:

$$\begin{aligned} |W| \geq \quad |A| \quad & + |D0| \times (|O1|/|A| - 1) + |D1| \times (|O2|/|O1| - 1) \\ & + |D2| \times (|O3|/|O2| - 1) + |D3| \times (|C|/|O3| - 1) \end{aligned}$$

This in turn can be bounded by

$$|W| \geq \min(|A|, |O1|, |O2|, |O3|, |C|)$$

, since $|D0| \leq |A|, |D1| \leq |O1|,$ etc. □

The tensors in the four-index transform contains permutation symmetry that can be represented as $A[ij, kl]$, $O1[\alpha, j, kl]$, $O2[\alpha\beta, kl]$, $O3[\alpha\beta, \gamma, l]$, and $C[\alpha\beta, \gamma\delta]$. While all tensors contain some degree of permutation symmetry, the output tensor C also contains spatial symmetry. Because this makes the output $|C|$ the smallest of all five tensors, we expect the necessary condition for the possibility of full reuse to be: $S \geq |C|$.

Theorem 6.2 (With Symmetry)**.** *Consider the four-index transform computation op*1234 *(Equation 2 with symmetries in Table 1). Then,*

$$IO_{opt}(op1234) = |A| + |C| \text{ only if } S \geq |C|$$

```
1  /* In this code a=alpha,b=beta,
2     c=gamma,d=delta. If S > |C|,
3     then total I/O is
4     |C|+|B1|+|B2|+|B3|+|B4|+|A| */
5  for l
6    malloc (O1,n^3); malloc(A_buf,n*3)
7    copy (A_buf, A[(i>j)*,k*,l])
8    for a, i, j, k
9      O1[a,j,k] += A_buf[i>j,k]*B1[a,i]
10   delete(A_buf); malloc(O2,n^3/2)
11   for a > b, j, k
12     O2[a>b,k] += O1[a,j,k]*B2[b,j]
13   delete (O1); malloc(O3, n^3/2)
14   for a>b, c, k
15     O3[a>b,c] += O2[a>b,k]*B3[c,k]
16   delete(O2)
17   for a>b, c>d
18     C[a>b,c>d] += O3[a>b,c]*B4[d,l]
```

Listing 7: Fused four-index transform that achieves I/O equal to the size of input and output

Proof. We note that Equation 6 gives the live-set for both symmetric and non-symmetric contractions. We then prove by contradiction that Equation 6 is greater than $|C|$. To make Equation 6 smaller than $|C|$, each term in Equation 6 must be smaller than $|C|$. This forces $|D0|$ to be greater or equal to $|A| - |C|$, i.e., $|D0| \geq 7|A|/8$ based on Tab. 1. Similar to the previous proof, we have that $D0_{j,kl}$ has at most j elements (exploiting the symmetry $j < i$) and that $F1_{j,k,l}$ has exactly n_α elements. Again, this leads to $|F1| \geq |D0| \times |O1|/|A|$ i.e., $|F1| \geq 7/8|O1| \geq |C|$. In general, enforcing the condition that each term must be smaller than $|C|$ forces Li on the next term to be much larger than $|C|$ for all terms except the last one, where $|F4|$ is forced to be $|C|$. This leads to a contradiction, proving that $S \geq |W| \geq |C|$. □

6.2 Sufficient Condition

Listing 7 shows a fused schedule for $op1234$ with I/O cost of $|A| + |C|$ when $S \geq |C|$, thus establishing that $|A| + |C|$ is a tight I/O bound for $op1234$. Keeping in mind that $S \geq |C|$ must be true, the key idea we used to obtain this schedule was to only consider schedules that retained the output C in fast memory through the entire computation.

In this schedule, we fuse loop `l` across all contractions. Then for each iteration of loop `l`, we compute $O1[*, *, *, l]$, $O2[*, *, *, l]$, and $O3[*, *, *, l]$. Each iteration of `l` thus requires $A[*, *, *l]$ as the input, and no two iterations of `l` depend on the same values of A, $O1$, $O2$, or $O3$. Thus values of the input and the intermediates for a particular iteration of `l` can be discarded before the next iteration as they are fully reused during the iteration. However, each iteration involves a partial contribution to each element of the output tensor C. Therefore, C must be kept in fast memory during all iterations of `l`.

This schedule achieves an I/O of $|A| + |C|$ if $S \geq |C| + 2n^3$, where the first term corresponds to the size of fast memory needed to keep C in fast memory across iterations of l, and the second term the capacity needed to retain intermediates in fast memory without any spilling. Thus, we have a schedule $op1234$ with $IO_{opt}(op1234) = |A| + |C|$,

when $S \geq |C|$ (ignoring the lower order term n^3, which is negligible compared to $|C|$).

7. A New Parallel Four-Index Transform

Based on our analysis of fusion for the four-index transform, we next develop a fused schedule for it with the following properties:
1. Largest zero-spill four-index transform: For a given aggregate physical memory capacity of the distributed memory system, the schedule allows for the execution of the largest possible four-index transform problem, without requiring any disk I/O, or redundant recomputation of atomic integrals.
2. Minimization of communication volume: The fused schedule minimizes the communication volume between distributed memory nodes, thus enhancing performance for systems where inter-processor communication overhead is a significant performance bottleneck.

7.1 Largest In-Memory Four-Index Transform

In a distributed-memory system, if we consider the disk-based file system to represent the unbounded slow memory, and the aggregate physical memory as the fast memory, we can apply the results from the previous sections that are based on a two-level abstraction of the memory hierarchy. From Theorem 6.2, the aggregate physical memory in the cluster must be larger than $|C|$, to achieve an I/O of $|A|+|C|$. Our implementation of the four-index transform is based on the Listing 7, which achieves this I/O when the size of global memory is greater than $|C|$. Because $S \geq |C|$ is a necessary condition for complete reuse of the intermediates, it is guaranteed that our solution will allow spill-free execution of the largest possible four-index transform for a given amount of aggregate physical memory in the cluster.

In addition, the I/O for our implementation between disk an global memory is actually zero. The I/O reduces from $|A| + |C|$ to zero because of two reasons:
1. The input tensor is produced on the fly, i.e., elements of it are computed without ever having to store the entire tensor. It is produced in fast memory as needed.
2. For all problems of practical interest, $S \geq |C|$ The output C fits in fast memory, and therefore it can be consumed by the next step in the calculation, without ever having to store it in disk.

Listing 8 shows the details of the implementation. Each loop iterates over tile indices. Loop `l` (line 2) is the fused loop over tile l. Iterations of this loop are executed sequentially, while the computation within this loop is parallelized. Each iteration of loop `l` computes an intermediate sub-tensor of size N^3. In other words, for a fixed value of `l`, a slice of N^3 elements of $O1$, $O2$, and $O3$ are produced. For example, lines 3..14 show the computation of a sub-tensor of $O1$. The sub-tensor is distributed and requires $\mathcal{O}(N^3)$ global memory. Each intermediate sub-tensor is used to compute the next intermediate sub-tensor. This computation is parallelized. Each processor only computes the tiles it owns.

```
1  int node_id = GA_Nodeid()
2  for l
3    GA_Create(A,n^3/2);
4    for i>j, k
5      bufA[i>j,k] = ComputeA(i>j,k,l)
6      GA_Put(bufA,A[i>j,k])
7
8    GA_Create(O1,tilewidth* n^3)
9    for j, k
10     if (node_id owns O1[alpha*,j,k])
11       bufA = GA_Get(A[i*,j,k,l])
12       for alpha
13         for i
14           bufB = ComputeB(alpha,i)
15           DGEMM(bufA[i],bufB,buffO1)
16         GA_Put(bufO1, O1[alpha,j,k])
17   GA_Sync()
18
19   GA_Create(O2,tilewidth*n^3/2)
20   for alpha, k
21     if (node_id owns O2[alpha,beta*,k])
22       bufO1 = GA_Get(O1[alpha,j*,k])
23       for beta //less than alpha
24         for j
25           bufB = ComputeB(beta,j)
26           DGEMM(bufO1[j],bufB,bufO2)
27         GA_Put(bufO2, O2[alpha>beta,k])
28   GA_Sync(); delete O1
29
30   GA_Create(O3,tilewidth*n^3/2)
31   for alpha>beta
32     if (node_id owns O3[alpha>beta,gamma*])
33       bufO2 = GA_Get(O2[alpha,beta,k*,l])
34       for gamma
35         for k
36           bufB = ComputeB(gamma,k)
37           DGEMM(bufO2[gamma],bufB,bufO3)
38         GA_Put(bufO3, O3[alpha>beta,gamma])
39   GA_Sync(); delete O2
40
41   for alpha>beta, gamma
42     if (node_id owns C[alpha>beta,gamma>delta])
43       bufO3 = GA_Get(O3[alpha>beta,gamma])
44       for delta //less than gamma
45         bufB = Compute(delta,l)
46         DGEMM(bufO3,bufB,buffC)
47         GA_Acc(buffC,C[alpha>beta,gamma>delta])
48   GA_Sync(); delete O3
```

Listing 8: Implementation that fuses all four contractions to allow execution of the largest possible problem size without recomputation

Finally, the intermediate sub-tensor $O3$ is used to compute partial contributions to output tensor C. Each iteration of loop `l` computes a partial contribution to the entire output tensor C. Therefore, C must fit in global memory.

The disk to memory communication volume for this implementation is 0, and the total global memory required is:

$$\frac{N_i N_j N_k T_l}{2} + \frac{N_\alpha N_\beta N_k T_l}{2} + \frac{N_\alpha N_\beta N_\gamma N_\delta}{32} \tag{7}$$

7.2 Minimizing Communication Volume

I/O at the second level of the memory hierarchy—between the global (distributed) memory and the local memory—corresponds to communication between distributed memory nodes. At this level of the memory hierarchy, global memory is the unbounded slow memory, while local memory is the bounded fast memory. We can minimize the I/O between global memory and local memory by making the following two observations:

```
//if S > n^2, total I/O = |A|+|O2|+|O2|+|C|
for k > l
  for alpha
    for i, j
      O1[j] += A[i>j,k>l] * B[beta,i]
    for beta //less than alpha
      for j
        O2[alpha>beta,k>l] = O1[j] * B2[beta,j]
for alpha>beta
  for gamma
    for k, l
    O3[l] = O2[alpha>beta,k>l]* B3[gamma,k]
    for delta //less than gamma
      for l
        C[alpha>beta,gamma>delta] +=
          O3[gamma,l]*B4[delta,l]
```

Listing 9: Fused schedule op12/34

1. The computation within the fused loop in Listing 8 is itself a four-index transformation (referred to as the inner four-index transform). By fusing loop l, we have reduced the size of index l within the fused loop to either 1, in absence of tiling, or T_l in the presence of tiling, where T_l is the tile width. However, the fusion analysis presented in this paper still applies to the inner four-index transform.
2. For large problems of interest, the size of the local memory is smaller than the final output C of the inner four-index transform. Thus, full reuse of intermediates is not possible, based on Theorem 6.2. However, from Theorem 6.2 and Theorem 5.2, we know that schedule op12/34 (shown in Listing 9) achieves an I/O bound lower than all other fusion choices, when $S = \Omega(n^2)$ and $S < |C|$. Thus, $op12/34$ can be used to minimize communication volume for the inner four-index transform. Such an implementation is shown in Listing 10. In addition to fusing loop l, this implementation performs additional fusion of the first two and the last two contractions of the inner four-index transform. The k and $alpha$ loops are fused between the first two contractions, and the $alpha, beta$ and $gamma$ loops are fused between the second two contractions, similar to the $op12/34$ schedule shown in Listing 9.

The communication volume between global and local memory for this implementation is $|A| + |O2| + |O2| + |C|$, and the memory required is

$$\frac{N_i N_j N_k T_l}{2} + N_\alpha N_j N_k T_l + \frac{N_\alpha N_\beta N_k T_l}{2} + \frac{N_\alpha N_\beta N_\gamma T_l}{2} + \frac{N_\alpha N_\beta N_\gamma N_\delta}{32} \tag{8}$$

7.3 Mapping to Processors

On a conventional memory hierarchy with a single slow and fast memory, $op12/34$ eliminates I/O corresponding to tensors $O1$ and $O3$. To achieve the same effect on a parallel system with multiple fast memory (corresponding to local memory of individual processors), the mapping of computation to processors must be chosen carefully.

Consider the fused schedule for the first two inner contractions in Listing 10. Inside the fused loop k, each element of A is reused to compute multiple elements of O1 along $alpha$, and each element of $O1$ is reused to compute multi-

```
int node_id = GA_Nodeid() //this process's rank
for l
  GA_Create(A,n^3/2)
  for(i>j,k)
    bufA[i>j,k] = ComputeA(i>j,k,l)
    GA_Put(bufA,A[i>j,k])
  GA_Create(O2,tilewidth* n^3/2)
  for k
    if(node_id owns O2[(alpha>beta)*,k])
      bufA = GA_Get(A[(i>j)*,k])
      for alpha
        for j, i
          bufB = ComputeB(alpha,i)
          DGEMM(bufA[i>j],bufB,bufO1[j])

        for beta //less than alpha
          for j
            bufB = ComputeB(beta,j)
            DGEMM(bufO1[j],bufB,bufO2[beta])
          GA_Put(bufO2[beta], O2[alpha>beta,k])
  GA_Sync()
  for alpha > beta
    if(node_id owns C[alpha>beta,(gamma>delta)*])
      bufO2 = GA_Get(O2[alpha>beta,k*])
      for gamma, k
        bufB = ComputeB(gamma,k)
        DGEMM(bufO2[k],bufB,bufO3)

        for delta //less than gamma
          bufB = ComputeB(delta,l)
          DGEMM(bufO3,bufB,bufC)
          GA_Acc(bufC,C[alpha>beta,gamma>delta])
  GA_Sync(); delete O2
```

Listing 10: Implementation that performs outer fusion to maximize problem size and inner fusion to minimize communication.

ple elements of O2 along $beta$. If different values of $alpha$ (or $beta$) are mapped to different processors, then the same elements of A (or $O1$) must either be recomputed on each processor, or must be communicated. In order to avoid communication or redundant re-computation for A and $O1$, all values of $alpha$ and $beta$ for a given k and l must be computed on the same processor. As a result, only the fused loop k can be parallelized. Similarly, for the last two contractions, only the fused loops $alpha$ and $beta$ can be parallelized.

Parallelism vs communication+load imbalance: For large numbers of processors, there may not be enough parallelism in the inner fused schedule of the first two contractions, since only loop k can be parallelized. We overcome this problem in our implementation by allowing parallelization of $alpha$, which results in increased communication for A, by a factor equal to the parallelization of $alpha$. As this is also true for the unfused inner version, there is still a significant reduction in communication volume due to elimination of data movement for $O1$ and $O3$.

Another side effect of parallelization of $alpha$ is a potential for load imbalance. The load imbalance is due to permutation symmetry between $alpha$ and $beta$. As $alpha \geq beta$, a processor computing $alpha = 2$ only computes for two values of $beta$, while a processor computing for $alpha = 3$ computes three values of $beta$. Thus, there is potential for load imbalance. There are alternative load balancing strategies or strategies to increase parallelism, such

as block cyclic-distribution of tensors, or nested tiling of l, allowing loop k and parts of loop l to be parallelized.

7.4 To Fuse or not to Fuse?

When there is enough global memory to store the intermediates of the 4-index transform, the unfused implementation is better than the fused implementation. There are two reasons for this: i) As described in the previous section, the fused schedule is not perfectly load balanced; ii) Fusion leads to a space-time trade-off due to symmetry breaking.

In the absence of any permutation symmetry, the fused four-index transform and the unfused one have the same number of operations (additions and multiplications). However, in the presence of permutation symmetry, fusing loop l across all four contraction in the four-index transform requires breaking the symmetry between k and l indices, which doubles the number of computations for producing O1 and O2. Thus, our fused implementation performs approximately 1.5x more computation than the unfused schedule.

To optimize performance, we choose between the fused and the unfused implementations based on the available global memory. If the problem size is too large for the intermediates to fit in the aggregate physical memory of a system, we always chose the fused implementation. On the other hand, if the problem size is small enough for the intermediates to fit in global memory, we always choose the unfused implementation. We call this the fuse/unfuse hybrid implementation.

8. Experimental Results

In this section, we compare the performance of our fuse/unfuse hybrid implementation (described in Section 7.4) with the 4-index transform implementations in NWChem, for multiple problems sizes ranging from small to very large. We present results on three different distributed memory clusters, each with different characteristics. For each problem on each system, we compare our implementation to the fastest NWChem implementation for that problem on that system.

Platforms: *System A* is a small Infiniband cluster, where each node has two 4-core 2.53 GHz Intel Xeon E5630 (Westmere) processors and 24 GB main memory per node, connected as a full fat tree by QDR Infiniband (40 Gbps). *System B* is another small Infiniband cluster with 18 large memory nodes, where each node has two 14-core Intel Xeon E5-2680v4 2.4GHz processors and 512 GB main memory per node. *System C* is a large supercomputer interconnected via FDR Infiniband (14Gbps), with 1440 2.6GHz dual-socket Intel Xeon E5-2670 processors (8 cores per socket) and 128GB of main memory per node.

We used NWChem 6.5 compiled with GCC 4.4.7 on System A, NWChem 6.6 compiled with GCC 4.8 on System B, and NWChem 6.5 compiled with Intel ICC Compiler on System C.

Benchmarks: We evaluate our implementation using 5 different molecules of varying sizes: Hyperpolar, C60H20, Uracil, C40H56, and Shell-Mixed, consisting of 368 (small), 580 (medium), 698 (large), 1023 (very large) and 1194 (very large) orbitals, respectively. The number of orbitals corresponds to the size of each dimension of the 4-index transform. To compute these 4-index transforms in double precision without fusion requires at least 110 GB, 678 GB, 1.4 TB, 6.5 TB, and 12.1 TB of aggregate physical memory, respectively.

We ran the small, medium benchmarks only on small clusters, System A and System B. We ran the large benchmark on all three systems, and the very large benchmarks only on System B and System C, since only these two systems have sufficient aggregate physical memory to hold the final molecular integral.

Results and Discussion: Figure 2 presents experimental results comparing the fuse/unfuse hybrid implementation to the fastest NWChem implementation. For all problem on all systems, fuse/unfuse hybrid is as fast or faster than the *NWChem Best*, the fastest among available schemes in NWChem that could perform the integral transformation..

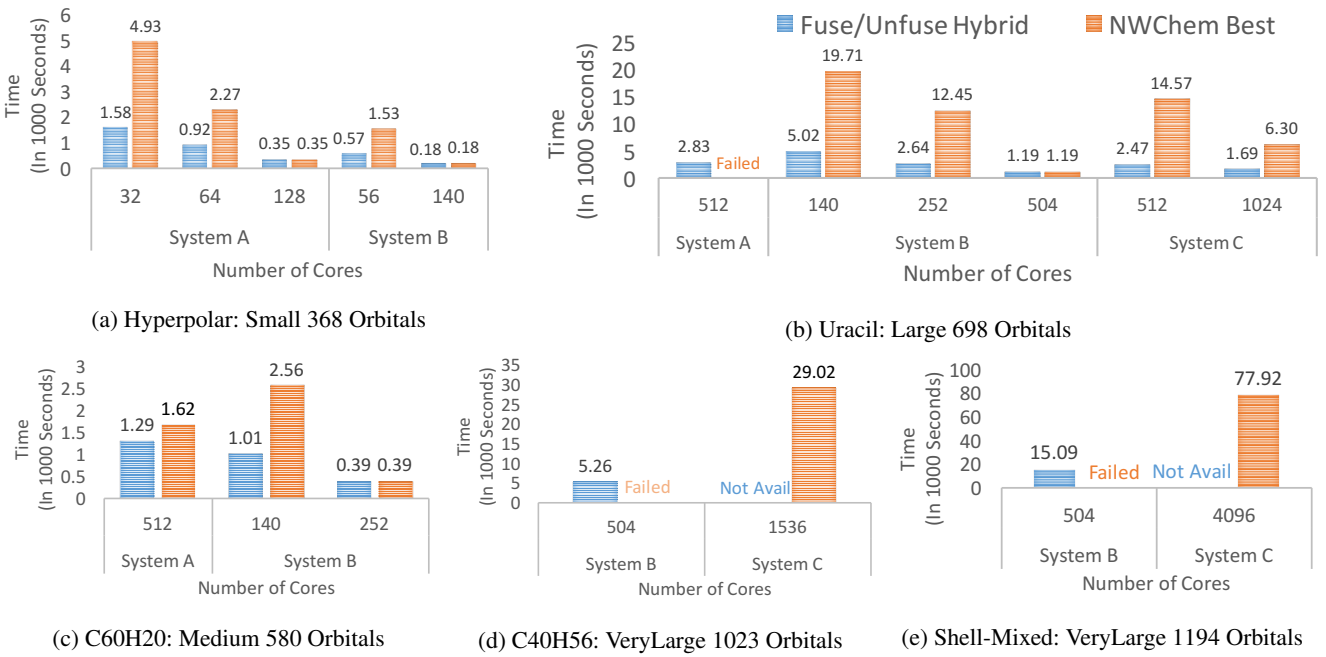
When there is insufficient aggregate physical memory to store the intermediates, our fuse/unfuse hybrid strategy uses the fused implementation which is 1.2x-6x faster than NWChem Best. This is demonstrated by i) small benchmark on 32 and 64 cores (4/8 nodes) of System A, and 56 cores (2 nodes) of System B, ii) medium benchmark on 140 cores (5 nodes) of System B, and iii) large benchmark on 140/252 cores (5/9 nodes) of System B and 512/1024 cores (128/256 nodes with 4 ranks per node) of System C.

When there is sufficient aggregate physical memory to store the intermediates, our fuse/unfuse hybrid strategy is as good as NWChem Best because both use the unfused implementation. This is demonstrated by i) small benchmark on 128 cores of System A and 140 cores of System B, ii) medium benchmark on 252 core of System B, and iii) large benchmark on 504 cores (18 nodes) of System B.

For a given amount of aggregate physical memory, our fused implementation can run much larger problems than possible with NWChem implementations. This is demonstrated by the large benchmark on 512 cores of System A, and by the very large benchmarks on 504 cores of System B. In each of these cases, the available aggregate physical memory was insufficient for any of the NWChem implementations to run (marked as "Failed"). However, our fused implementation ran successfully.

9. Conclusions

In this paper, we have addressed the development of a high-performance parallel implementation of the four-index integral transformation, a compute-intensive calculation used in many quantum chemistry models. Loop fusion and tiling are key optimizations, but the space of possible configurations is extremely high. Several implementations have been developed over the last two decades for the four-index transformation in NWChem, successively attempting to improve



(a) Hyperpolar: Small 368 Orbitals

(b) Uracil: Large 698 Orbitals

(c) C60H20: Medium 580 Orbitals

(d) C40H56: VeryLarge 1023 Orbitals

(e) Shell-Mixed: VeryLarge 1194 Orbitals

Figure 2: This figure shows the execution time for the 4-index transform, for problems of various sizes, on three different distributed memory clusters. It shows a comparison between the new implementation and the best NWChem implementation for the particular problem. Some problems ran out of memory on some systems using all NWChem implementations. Those are marked as failed. On System C, we were not able to acquire resources to run with our implementation for very large problems. Those are marked as Not Avail. However, we show the NWChem results obtained on System C to show the comparsion with the much smaller System B.

on previous versions. In this paper, we pursued a radically different approach to addressing the problem of determining which of many fusion configurations we should consider, using *data movement lower bounds* for analysis. The lower-bounds-based analysis enabled the development of an efficient parallel implementation that was demonstrated to be significantly better than the best current option in the production NWChem code.

Acknowledgment

We thank the reviewers of the paper for their valuable feedback and recommendations. This work was supported in part by DOE award DE-SC0012489, and NSF awards ACI-1440749 and ACI-1404995. Acess to parallel computer resources at the Ohio Supercomputer Center, CSE Department at Ohio State University, and Pacific Northwest national Laboratory are gratefully acknowledged.

A. Appendix: Proof of Fusion Lemma

We use the computational model of the red/blue pebble game [13] to prove the Fusion Lemma. The red/blue pebble game is defined over a computational directed acyclic graph (CDAG), where operations are represented as graph vertices and the flow of values between operations is captured by graph edges.

Definition A.1 (CDAG [7, 13])**.**
A computational directed acyclic graph (CDAG) is a 4-tuple $C = (I, V, E, O)$ *of finite sets such that: (1)* $I \subset V$ *is the input set and all its vertices have no incoming edges; (2)* $E \subseteq V \times V$ *is the set of edges; (3)* $G = (V, E)$ *is a directed acyclic graph; (4)* $V \setminus I$ *is called the operation set and all its vertices have one or more incoming edges; (5)* $O \subseteq V$ *is called the output set.*

The red-blue pebble game uses two kinds of pebbles: a fixed number (S) of red pebbles that represent small fast local memory (could represent cache, registers, etc.), and an unbounded number of blue pebbles that represent the large slow main memory. Starting with blue pebbles on all inputs nodes in the CDAG, the game involves the generation of a sequence of steps to finally produce blue pebbles on all outputs. The rules of the game are as follows.

Definition A.2 (Red-Blue pebble game (no repebbling))**.**
Given a CDAG $C = (I, V, E, O)$*, S red pebbles and unbounded number of blue pebbles, with a blue pebble on each input vertex, a complete calculation is any sequence of steps using the following rules that results in a final state with blue pebbles on all output vertices:*

R1 (Load) *A red pebble may be placed on any vertex that has a blue pebble (load from slow to fast memory),*

R2 (Store) *A blue pebble may be placed on any vertex that has a red pebble (store from fast to slow memory),*

R3 (Compute) *If all immediate predecessors of a vertex of* $V \setminus I$ *have red pebbles, if not already computed in the*

past, a red pebble may be placed on that vertex (execution or "firing" of operation),

R4 (Delete) *A red pebble may be removed from any vertex (reuse storage).*

The number of I/O operations for any complete calculation is the total number of moves using rules R1 or R2. We note that although the computational model of the red/blue pebble game abstracts away many features of real hardware, such as cache replacement policy, associativity, non-unit line size, etc., lower bounds established using the formalism of the red/blue pebble game are valid assertions on the minimal number of data elements that must be moved in the memory hierarchy of any real computer system.

Lemma A.3 (Fusion Lemma)**.** *Let computation $C1$ with CDAG $(V1, E1, I1, O1)$ and computation $C2$ with CDAG $(V2, E2, I2, O2)$ represent a producer-consumer pair, where $I2 \cap V1 = O1$. Let $IO_{LB}(C1)$ and $IO_{LB}(C2)$ represent known I/O lower bounds for $C1$ and $C2$, respectively. Consider the fused computation $C12$ fusing $C1$ and $C2$, with CDAG $(V12, E1 \cup E2, I1 \cup I2 - O1, O2)$, where $V12$ represents the union of vertices in $C1$ and $C2$, with each output vertex in $C1$ being merged with some input vertex of $C2$. Then, any valid schedule for $C12$ has an I/O lower bound given by: $IO_{LB}(C12) = IO_{LB}(C1) + IO_{LB}(C2) - 2 * |O1|$.*

Proof. We consider any valid schedule $S12$ for $C12$ and prove that $IO(S12) \geq IO_{LB}(C1) + IO_{LB}(C2) - 2 * |O1|$. 5The schedule $S12$ comprises of some sequence of the four kinds of actions described above. From this schedule $S12$, we first generate a valid augmented schedule $S12^+$ by insertion of additional Loads (rule $R1$) and Stores (rule $R2$) as follows:

Consider each vertex v in $C12$ that represents the merge of an output vertex in $C1$ and an input vertex in $C2$.

- Immediately after the (unique) Compute operation on vertex v, first insert a Store operation to place a blue pebble on the vertex.
- Let o be the earliest Compute operation in the augmented schedule $S12^+$ that uses the value produced in vertex v. Insert a Delete operation for v, followed by a Load operation for v immediately preceding the Compute operation o. This is valid since i) the operation o would have used the value of vertex v in a red pebble in $S12$, so that the Delete operation is valid and also frees a red pebble, and ii) $S12^+$ has a previously inserted Store operations for vertex v, as described above, so that a blue pebble is on v, and a freed red pebble is available to place on v.

The augmented schedule $S12^+$ is a valid schedule for $C12$ since each of the added operations is valid and the relative order of all Compute operations is unchanged from $S12$, i.e., no dependences are violated. From the valid augmented schedule $S12^+$, we construct valid schedules $S1$ and $S2$, respectively, for the unfused computations $C1$ and $C2$.

First, tag as belonging to $S1$, each operation in $S12^+$ on any vertices of $C12$ that correspond to $V1 - O1$. Additionally tag as belonging to $S1$, the Compute operations on vertices belonging to $O1$, as well as the inserted Store operations in $S12^+$ that immediately follow the corresponding Compute operations. By deleting all untagged operations in $S12^+$, the remaining operations form a valid schedule for $C1$. This is because (by induction on the steps of $S1$):

- Every vertex in $C1$ is included in $C12$, and $S12$ (and therefore also $S12^+$) must contain a Compute operation for each such vertex; each such Compute operation has valid operands in $S1$ since any predecessor vertices must belong to $C1$, and all operations on them from $S12^+$ will also appear in $S1$ before it.
- Similarly, every Load, Store, and Delete operation will have valid operands in $S1$ because the relative order of all operations in $S1$ is the same as that in $S12$, which is a valid schedule.
- There will never be any resource violation, i.e., unavailability of a red pebble for any Load or Compute operation because the number of free red pebbles just prior to execution of any operation in $S1$ is greater than or equal to the number of free red pebbles immediately before execution of the corresponding operation in $S12$. This is true at the start (all red pebbles are free) and as we execute operations, the number of free pebbles in $S12$ may go lower due to interleaved operations from vertices in $C2$ but can never be higher than that for $S1$.
- All outputs $O1$ are in blue pebbles at the end of $S1$, due to the explicitly inserted Store operations in construction of $S12^+$.

To construct a valid schedule $S2$ for $C2$, we first remove all $S1$-tagged operations from $S12^+$. We then remove the inserted Delete operations from the Delete/Load pair that was inserted before every first use of a vertex in $O1$ when constructing $S12^+$, as described earlier. This removes all operations pertaining to computation of $C1$ vertices, but if we assume all inputs $I2$ are in blue pebbles before the start of the schedule $S2$, the necessary load operations are all available in $S2$ - for those elements of $I2$ that are in $O1$, explicit Load operations were inserted in constructing $S12^+$, and for other elements of $I2$, $S12$ must already have included the needed Load operations. A similar reasoning as detailed for $S1$ can be carried out for $S2$ to show that each individual operation in $S2$ will not violate any dependences (relative order of operations in $S2$ is the same as $S12$) and that no resource violations can occur, proving that $S2$ is a valid schedule for $C2$.

We now have:

- $IO(S12^+) = IO(S1) + IO(S2)$ (because $S1$ and $S2$ are obtained by a disjoint partition of operations in $S12^+$).
- $IO(S12^+) = IO(S12) + 2 * |O1|$ (because in constructing $S12^+$ from $S12$, exactly $|O1|$ Store operations and $|O1|$ Load operations were inserted, as described above).
- $IO(S1) \geq IO_{LB}(C1)$ (any valid actual schedule must have I/O cost greater than or equal to any valid lower bound for the computation).
- $IO(S2) \geq IO_{LB}(C2)$.

Hence, $IO(S12) + 2 * |O1| \geq IO_{LB}(C1) + IO_{LB}(C2)$, i.e., $IO(S12) \geq IO_{LB}(C1) + IO_{LB}(C2) - 2 * |O1|$.

References

[1] ACES II, a program product of the quantum theory project. *See http://www.qtp.ufl.edu/aces/*, 1996.

[2] The massively parallel quantum chemistry program (MPQC). *http://www.mpqc.org/index.php*, 2004.

[3] MOLPRO, a package of ab initio programs. *See http://www. molpro. net*, 2006.

[4] Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *See http://www. nwchem-sw. org/index. php*, 2010.

[5] Psi4, an open-source ab initio electronic structure program. *See http://www.psicode.org/*, 2012.

[6] M. Abe, T. Yanai, T. Nakajima, and K. Hirao. A four-index transformation in dirac's four-component relativistic theory. *Chem. Phys. Letters*, 388(13):68 – 73, 2004.

[7] G. Bilardi and E. Peserico. A characterization of temporal locality and its portability across memory hierarchies. *Automata, Languages and Programming*, pages 128–139, 2001.

[8] L. A. Covick and K. M. Sando. Four-index transformation on distributed-memory parallel computers. *J. Comp. Chem.*, 11 (10):1151–1159, 1990.

[9] J. Dongarra, J.-F. Pineau, Y. Robert, and F. Vivien. Matrix product on heterogeneous master-worker platforms. In *PPoPP*, pages 53–62, 2008.

[10] G. Fletcher, M. Schmidt, and M. Gordon. Developments in parallel electronic structure theory. *Adv. Chem. Phys.*, 110: 267–294, 1999.

[11] T. R. Furlani and H. F. King. Implementation of a parallel direct scf algorithm on distributed memory computers. *J. Comp. Chem.*, 16(1):91–104, 1995.

[12] X. Gao, S. Krishnamoorthy, S. K. Sahoo, C. Lam, G. Baumgartner, J. Ramanujam, and P. Sadayappan. Efficient search-space pruning for integrated fusion and tiling transformations. *CCPE*, 19(18):2425–2443, 2007.

[13] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *STOC*, pages 326–333, 1981.

[14] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[15] C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Comp. Lang. Sys. Struc.*, 37(2): 63–75.

[16] A. C. Limaye and S. R. Gadre. A general parallel solution to the integral transformation and second-order Møller-Plesset energy evaluation on distributed memory parallel machines. *J. Chem. Phys.*, 100(2):1303–1307, 1994.

[17] W. Ma, S. Krishnamoorthy, and G. Agrawal. Practical loop transformations for tensor contraction expressions on multi-level memory hierarchies. In *CC 2011*, pages 266–285, 2011.

[18] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[19] M. Pernpointner, L. Visscher, W. A. de Jong, and R. Broer. Parallelization of four-component calculations. i. integral generation, SCF, and four-index transformation in the Dirac-Fock package MOLFDIR. *J. Comp. Chem.*, 21(13): 1176–1186.

[20] G. Rauhut, P. Pulay, and H.-J. Werner. Integral transformation with low-order scaling for large local second-order Møller-Plesset calculations. *J. Comp. Chem.*, 19(11):1241–1254.

[21] S. Sæbø and J. Almlöf. Avoiding the integral storage bottleneck in LCAO calculations of electron correlation. *Chem. Phys. Let.*, 154(1):83 – 89, 1989.

[22] S. K. Sahoo, S. Krishnamoorthy, R. Panuganti, and P. Sadayappan. Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In *SC 2005*.

[23] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, et al. General atomic and molecular electronic structure system. *J. Comp. Chem.*, 14(11): 1347–1363, 1993.

[24] R. A. Whiteside, J. S. Binkley, M. E. Colvin, and H. F. Schaefer III. Parallel algorithms for quantum chemistry. i. integral transformations on a hypercube multiprocessor. *J. Chem. Phys.*, 86(4):2185–2193, 1987.

[25] S. Wilson. Four-index transformations. In *Methods in Computational Chemistry*, pages 251–309. Springer, 1987.

[26] T. L. Windus, M. W. Schmidt, and M. S. Gordon. Parallel algorithm for integral transformations and GUGA MCSCF. *Theoretica chimica acta*, 89(1):77–88, 1994.

[27] A. T. Wong, R. J. Harrison, and A. P. Rendell. Parallel direct four-index transformations. *Th. Chim. Acta*, 93(6):317–331.